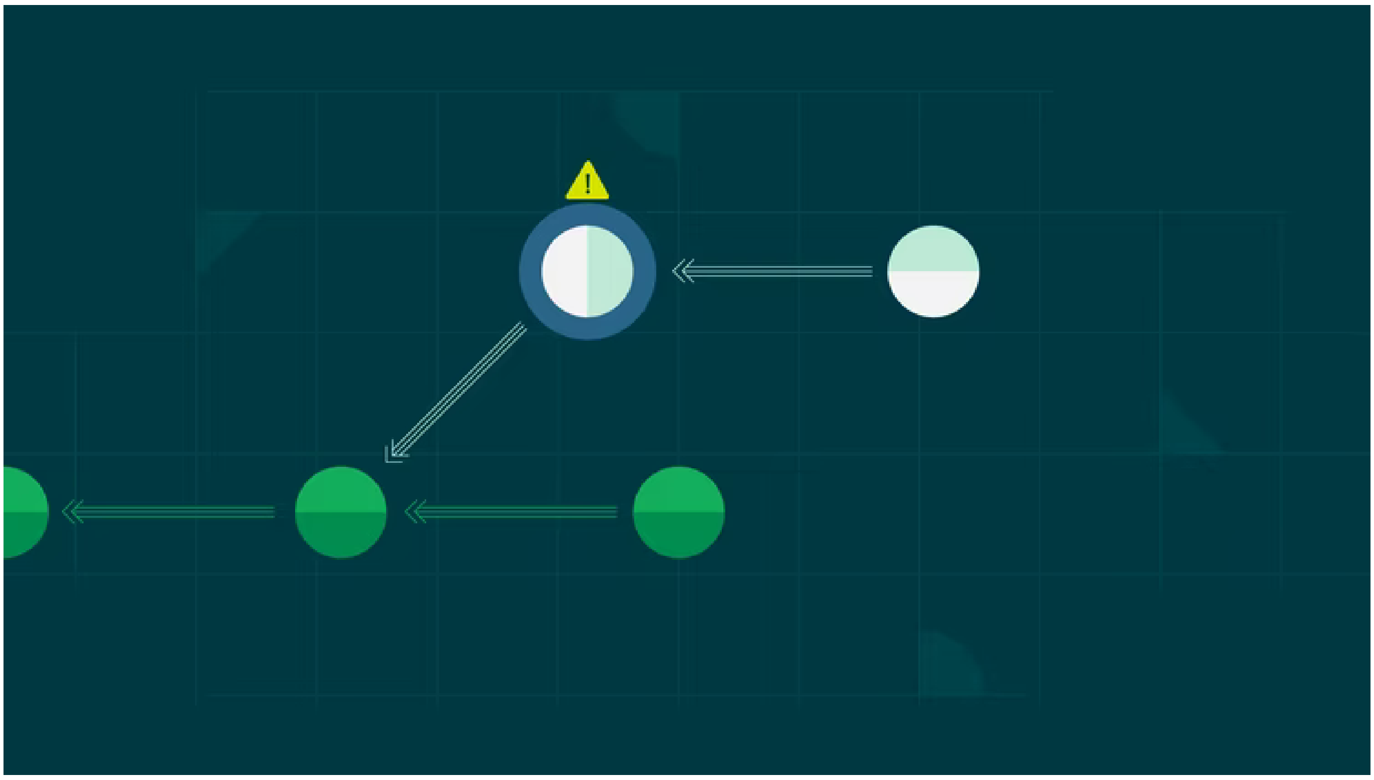
Recovering from the Git detached HEAD state

Ron Powell
Manager, Marketing Insights and Strategy



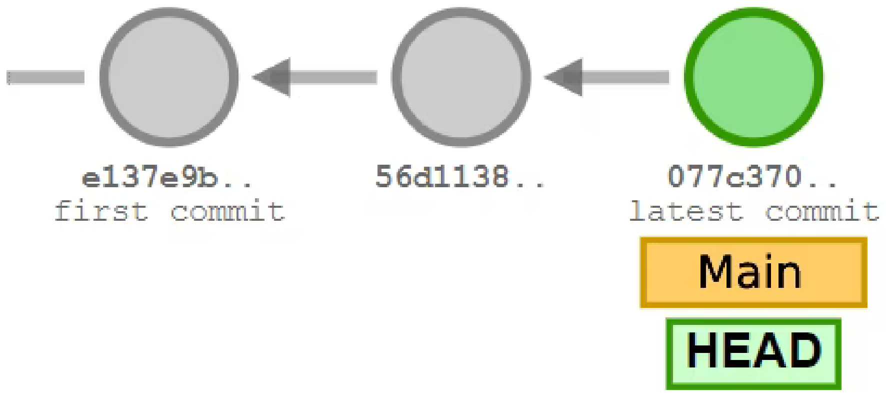
The introduction of Git as a source-code management system in 2005 fundamentally transformed the process of software development. Git allows developers to maintain a history of changes, or commits, to their code and revert to previous commits in seconds if something goes wrong. It makes collaboration easier by allowing branching to keep code for different features separate and seamlessly merging commits by various people. It is fast, scalable, and has more commands and flexibility than older version control tools like Apache Subversion (SVN) and Concurrent Versions System (CVS).

However, learning Git is more complicated than learning SVN or CVS. Complex commands and a less intuitive user interface can sometimes lead to unwanted states, including a state called detached HEAD. In this article, we will explore what the Git detached HEAD state is and some situations that cause it. Then, we will demonstrate how to save or discard changes in a detached head so you can quickly recover from the situation.

What does detached HEAD mean?

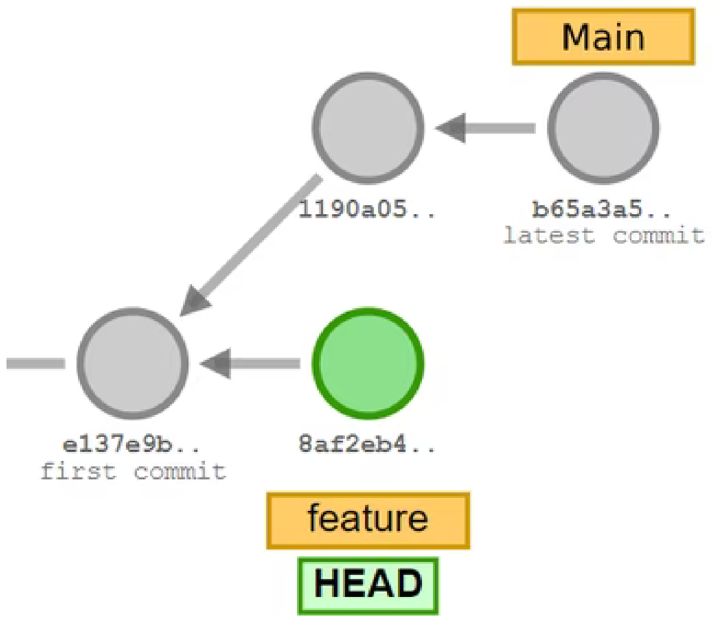
In Git, HEAD refers to the currently checked-out branch’ s latest commit. However, in a detached HEAD state, the HEAD does not point to any branch, but a specific commit or the remote repository.

Below is a diagram of the Git HEAD in a normal state, pointing to the latest commit in the main branch.



In this image, the HEAD points to the latest commit and current checked-out branch on every commit in the current branch.

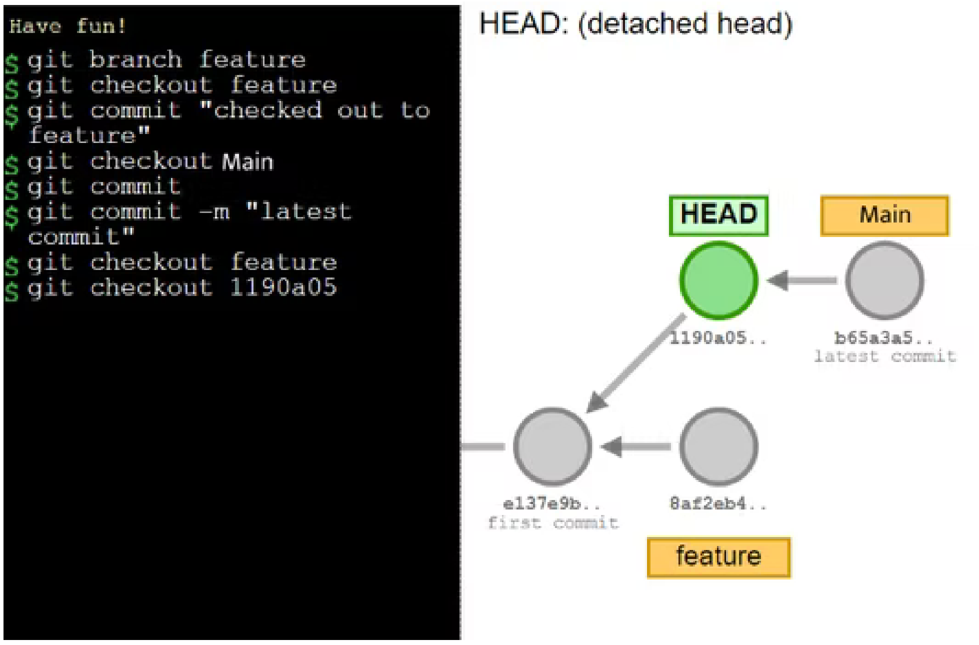
HEAD is also a common state when working with multiple branches.



In this situation, you have two branches, the main branch and a feature branch. Since you are checked out to the feature branch, HEAD points there. You can create this scenario with the following Git commands:

```
git branch feature
git checkout feature
git commit -m “checked out to feature”
git checkout Main
git commit
git commit -m “Latest commit”
git checkout feature
```

In both of the diagrams above, HEAD points to the most recent commit in the currently checked out branch, which is its normal state. In Git, you can also check out a particular commit, which results in the detached HEAD state. Continuing from the previous scenario, if you check out to the first commit on the main branch, you will see that HEAD is now detached.



```
Have fun!
$ git branch feature
$ git checkout feature
$ git commit "checked out to
  feature"
$ git checkout Main
$ git commit
$ git commit -m "latest
  commit"
$ git checkout feature
$ git checkout 1190a05
```

In this case, HEAD does not point to any branch—it references a commit.

Scenarios that can cause a detached HEAD state

You can find yourself in a detached HEAD state primarily through two scenarios:

1. Checking out a specific Secure Hash Algorithm 1 (SHA-1) commit hash
2. Checking out to a remote branch without fetching it first

We already demonstrated that if you check out the SHA-1 commit hash, you will be in the detached HEAD state. Another situation that causes a detached HEAD is checking out the remote branch. If you check out to the origin (main) branch, which is read-only, you will be in the detached HEAD state.

Some other scenarios can cause a detached HEAD as well. For example, checking out to a specific tag name or adding `^0` on any given branch causes the detached HEAD state.

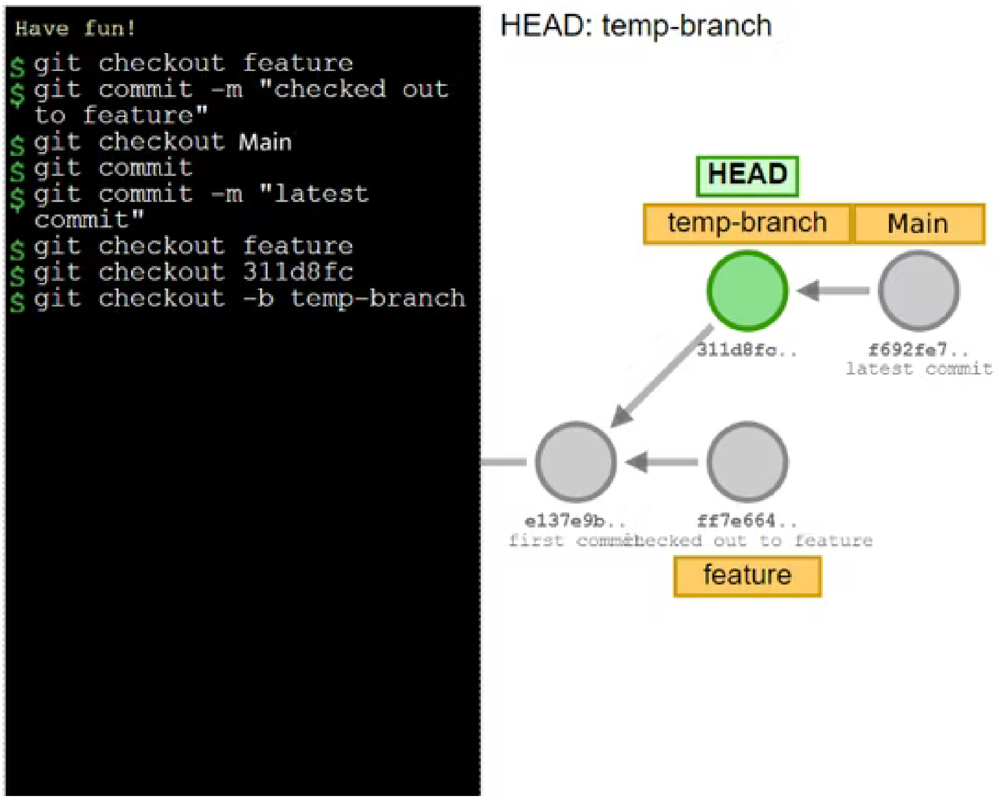
How to save changes in a detached HEAD

If you find yourself a detached HEAD state and realize it quickly, you can quickly recover by checking out the previous branch. But what if you are in the detached HEAD state by mistake then perform commits over commits? If you are committing in the detached HEAD state, does that mean your changes are not saved?

Not at all. It just means you are not currently attached to any branch, and as a result, your HEAD is detached. If you want to keep the changes you made while in the detached HEAD state, you can solve this problem with three simple steps: creating a new branch, committing the changes, and merging the changes.

Create a new branch

To save changes committed in a detached HEAD state, you first need to create a new branch.

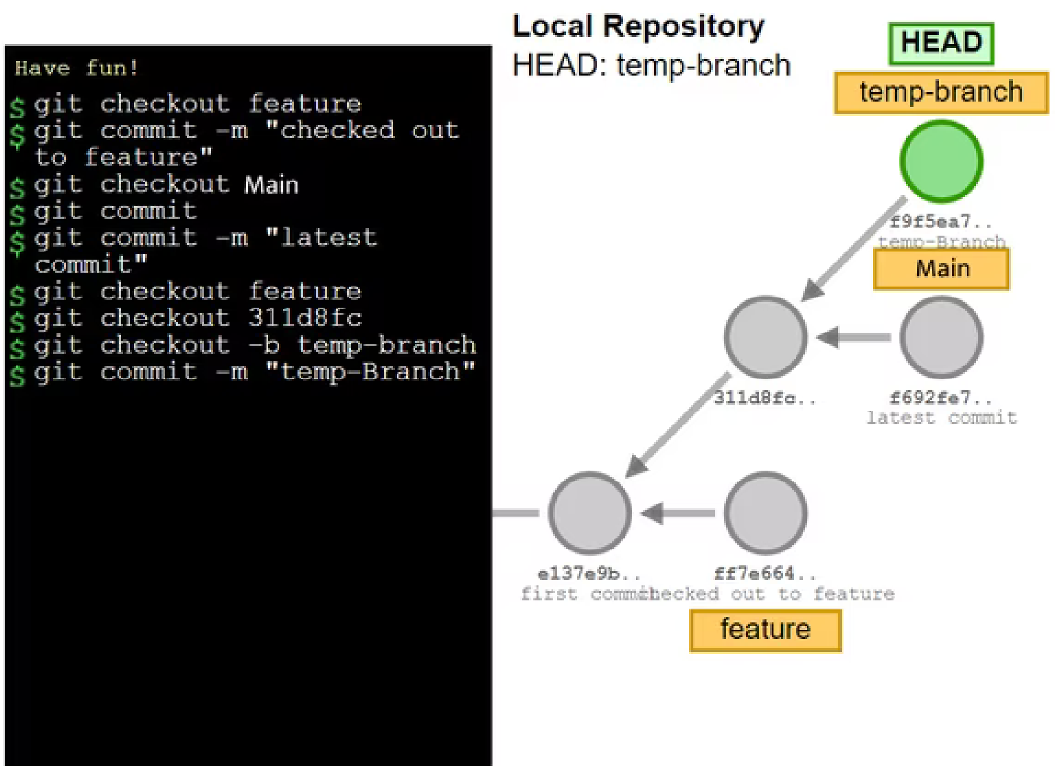


```
Have fun!
$ git checkout feature
$ git commit -m "checked out
  to feature"
$ git checkout Main
$ git commit
$ git commit -m "latest
  commit"
$ git checkout feature
$ git checkout 311d8fc
$ git checkout -b temp-branch
```

Continuing from the scenario described above, you create a new branch called `temp-branch`. As soon as you make the branch and check out to it, the HEAD is no longer detached.

Commit the changes

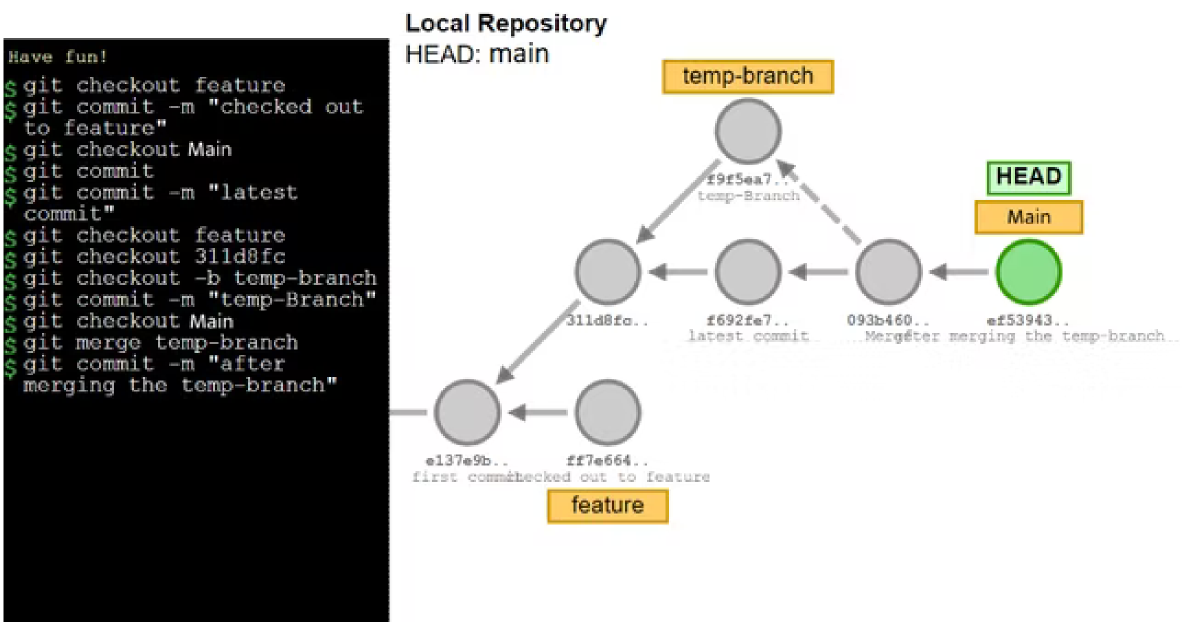
After checking out to the new branch, you can commit the changes, which Git will preserve.



```
Have fun!
$ git checkout feature
$ git commit -m "checked out
  to feature"
$ git checkout Main
$ git commit
$ git commit -m "latest
  commit"
$ git checkout feature
$ git checkout 311d8fc
$ git checkout -b temp-branch
$ git commit -m "temp-Branch"
```

Merge the changes

Now, you check out to the branch where you want the changes. In this case, you want to merge the changes to the main branch. So, you first need to check out the main branch, then merge the changes from `temp-branch` and add the final commit message.



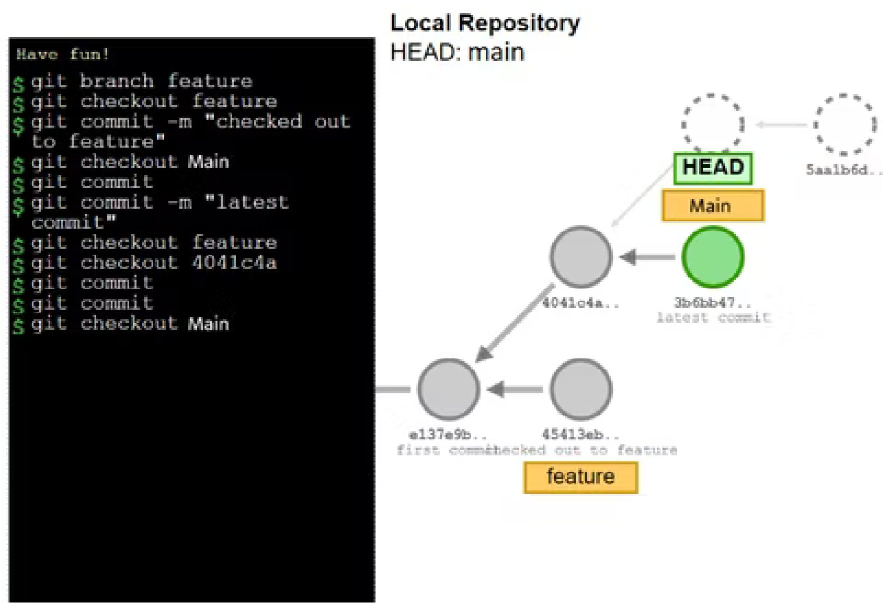
```
Have fun!
$ git checkout feature
$ git commit -m "checked out
  to feature"
$ git checkout Main
$ git commit
$ git commit -m "latest
  commit"
$ git checkout feature
$ git checkout 311d8fc
$ git checkout -b temp-branch
$ git commit -m "temp-Branch"
$ git checkout Main
$ git merge temp-branch
$ git commit -m "after
  merging the temp-branch"
```

With these simple steps, you have successfully preserved your changes and recovered from the Git detached HEAD state.

How to discard changes in a detached HEAD

If you want to discard the changes in the detached HEAD state, you only need to check out to the existing or previous branch. The commits on the detached HEAD state will not affect your existing branch, and Git will archive them.

The diagram below shows a situation in which, after going into the detached HEAD state, you make two commits that you do not want to keep. Then, you check out to the main branch. The dotted circles indicate that these commits are no longer part of any branch, and Git will delete them.



```
Have fun!
$ git branch feature
$ git checkout feature
$ git commit -m "checked out
  to feature"
$ git checkout Main
$ git commit
$ git commit -m "latest
  commit"
$ git checkout feature
$ git checkout 4041c4a
$ git commit
$ git commit
$ git checkout Main
```

Note that once Git prunes your detached HEAD state commits, there is no way to get them back. However, if they have not been deleted, you can check out to that SHA-1 commit hash, create a branch, and merge it to the desired branch to preserve the changes.

Conclusion

Git is a valuable developer tool and more popular than older versioning tools such as CVS and Subversion. That said, it can be more complex and challenging to master, and can sometimes lead to confusing situations such as the detached HEAD state.

If you find yourself in the detached HEAD state, remember that you can always preserve your changes by creating and checking out to a new branch, then committing and merging the changes in the desired branch. If you do not want to save the changes, you can simply check out to any branch, and Git removes those commits.

Also, Git 2.23 has a new command, `git switch`. This is not a new feature but an alternative command to `git checkout` so you can switch between the branches and create a new branch. To change from one branch to another, use `git switch branchName` to create a new branch, then switch to it using the `git switch -c branchName` command.

Although finding your code in the detached HEAD state is not ideal, you can use these methods to move or remove your commits and quickly get your project back on track.